2010

MPI Enhancements in John the Ripper

Edward R. Sykes
Sheridan College, ed.sykes@sheridancollege.ca

Michael Lin
Sheridan College

Wesley Skoczen
Sheridan College, wesley.skoczen@sheridancollege.ca

MPI Enhancements in John the Ripper

Edward R. Sykes, Michael Lin, Wesley Skoczen
Sheridan Institute of Technology and Advanced Learning, 1430 Trafalgar Road, Oakville, Ontario, L6H 2L1, Canada

E-mail: {ed.sykes; michael.lin; wesley.skoczen}@sheridanc.on.ca

Abstract. John the Ripper (JtR) is an open source software package commonly used by system administrators to enforce password policy. JtR is designed to attack (i.e., crack) passwords encrypted in a wide variety of commonly used formats. While parallel implementations of JtR exist, there are several limitations to them. This research reports on two distinct algorithms that enhance this password cracking tool using the Message Passing Interface. The first algorithm is a novel approach that uses numerous processors to crack one password by using an innovative approach to workload distribution. In this algorithm the candidate password is distributed to all participating processors and the word list is divided based on probability so that each processor has the same likelihood of cracking the password while eliminating overlapping operations. The second algorithm developed in this research involves dividing the passwords within a password file equally amongst available processors while ensuring load-balanced and fault-tolerant behavior. This paper describes John the Ripper, the design of these two algorithms and preliminary results. Given the same amount of time, the original JtR can crack 29 passwords, whereas our algorithms 1 and 2 can crack an additional 35 and 45 passwords respectively.

1. Introduction

The password is the most common method of authentication in computing. Used in both low and high security applications, alone or in combination with other authentication methods, the character password remains essential to modern computer security. While the level of security provided by password based authentication depends on a wide variety of factors involving how the password is handled, stored and used, the strength of the password itself depends only on its length and formation.

Password strength is a measure of a password's resistance to guessing. Assuming all other factors of a password based authentication system are immune to compromise, a password may still be guessed. Therefore ensuring that all passwords in a system are of a reasonable strength to resist most guessing attempts is essential, and thus is the purpose of password policy.

John the Ripper (JtR) is an open source software package commonly used by system administrators to enforce password policy. JtR is designed to attack (i.e., crack) passwords encrypted in a wide variety of commonly used formats. It is most commonly used to recover passwords in a computer system and is a useful tool for detecting weak passwords. JtR also has limited application in approximating the abilities of would be attackers whose tools may operate in a similar way.

JtR is a serial program that does not possess native support for parallelization. While cracking short or simple passwords is feasible, stronger passwords are very computationally expensive to crack. Each crack attempt is a reasonably discrete operation that does not directly benefit from previous or future attempts. This makes JtR a prime candidate for parallelization.

Several attempts have been made to parallelize JtR's brute-force attack mode [1], and an attempt to parallelize the wordlist mode has also been made [2]. Furthermore, several script-based attempts to parallelize JtR by running multiple serial JtR processes in parallel have been made but with limited success.

In this research we are interested in the number of passwords that can be cracked in the finite amount of time because this is ultimately what the password-collector ultimately is interested in. We excluded passwords that are impossible to crack in the finite amount of time. In pursuing this goal, two distinct algorithms were designed, implemented and evaluated as enhancements to JtR using the Message Passing Interface. The first algorithm uses multiple processors to crack one password by using an innovative approach to workload distribution. In this algorithm the candidate password is distributed to all participating processors and the wordlist is divided based on probability so that each processor has the same likelihood in cracking the password. The overarching goal of this algorithm is to crack a single password as quickly as possible. The second algorithm is intended to be used by system administrators for systems in which thousands of accounts may be in use. This algorithm divides a potentially large password file equally amongst participating processors. This approach offers efficient load-balancing and fault-tolerant performance.

This paper discusses the single processor version of JtR in Section 2 and related work in Section 3. Section 4 presents the design of our two JtR algorithms. Section 5 presents the preliminary results followed by a discussion and future work.

2. John the Ripper

There are many approaches to cracking encrypted passwords. One possible approach might be to reverse engineer the encryption algorithm and recover the password algorithmically, which unfortunately is both exceedingly difficult and highly impractical. JtR cracks passwords by attempting to produce a text string that matches the original password by a process of *guessing*.

Passwords on computer systems are commonly stored together in files, along with other data like usernames and account information. To secure the password files their contents are encrypted, usually with a 1-way hashing algorithm. These algorithms produce an effectively unique summary, or message digest, of the original password. When the password hashes are used for authentication at a later time, the user inputted password is hashed and compared with the original hash stored in the password file. Since hashing algorithms produce effectively unique message digests, if the input was correct, the hashes will match. JtR takes advantage of this system of storage by using the same algorithms used to create the password files to encrypt a series of guesses, then the resulting hashes are compared with the originals.

Producing guesses for use in password cracking can be accomplished in a variety of ways. JtR uses versions of the brute-force and dictionary methods. The brute-force method is the most thorough and is theoretically infallible although extremely time consuming and computationally expensive. It simply exhaustively tries every possible character combination. JtR's implementation of the brute-force

method makes use of character frequency tables to try combinations containing more frequently used characters first.

The dictionary method makes use of a user definable list of possible passwords, a dictionary, as a guide for creating guesses. JtR provides a variety of options for applying the text in the dictionary. In addition to trying the dictionary text as is, salts can be added to pad the text and the dictionary text can also be mangled [3].

JtR has four modes of operation; incremental, single, wordlist, and external. The incremental mode simply applies the brute-force method while the wordlist and single modes use the dictionary method. The wordlist mode uses a user-definable text file as the dictionary while single mode generates its own dictionary using data found in the password file, such as the user name and account information. The external mode allows user defined modes of operation to be run using program code created with a subset of the C language.

A wide variety of options can be specified to customize JtR's operation. By default, if JtR is run without additional command line parameters, the single mode will be used first followed by the wordlist mode and finally the incremental mode.

JtR has been extended over the years to work with a wide variety of password file formats. The basic setup, without extensions, supports the following formats:

- Unix crypt(3) hashes:
  - traditional and double-length DES
  - BSDI extended DES
  - FreeBSD MD5
  - OpenBSD Blowfish
- Kerberos AFS
- Windows LM hash

However, JtR has some limitations. JtR's primary functional limitation is its restriction to operation on password files with known formats. It is also non-adaptive and cannot operate on file formats or hashing algorithms that aren't explicitly defined in its source code. Therefore JtR cannot be used directly on an arbitrary password authentication system.

JtR's primary performance limitation is its lack of support for parallelization. It can only make use of a single thread and does not support multi-thread operation natively.

3. MPI and John the Ripper

The current state of parallel implementation for JtR using Message Passing Interface (MPI) consists of a basic modification of the incremental mode. This implementation was initially created by Ryan Lim and later refined by multiple authors, notably John Anderson [1-4]. It uses MPI's facilities to divide the workload of the incremental mode by dividing the key space between the available computation processors.

The incremental mode's brute-force attack attempts every possible character combination in search of a hash that matches those listed in the password file. The MPI implementation essentially runs multiple serial incremental mode attacks, but each instance only works with a subset of the overall key space. The primary advantage to this approach is its simplicity and minimal alteration to the original

source code. Additionally, there is very minimal communication between processors, each processor calculates its own key space range and performs operations on it.

While this approach effectively parallelizes JtR's incremental mode, it has several limitations. The following list the current areas that are unsolved:

1. Due to minimal communication between processors, if a processor completes its allocated workload it will sit idle until all processors complete.
2. There is no rebalancing of the workload after the initial division.
3. This implementation also doesn't address issues which may arise from one or more processors going offline before execution completes.
4. Finally, this implementation only parallelized the incremental mode.

A separate attempt to parallelize JtR's wordlist mode using MPI has been made by Pippin et al [2]. This implementation is similar in basic concept to Lim's work with JtR's incremental mode in that it takes a minimalistic approach. MPI is used minimally and essentially multiple serial wordlist mode attacks are run, one per available computational processor.

One processor is chosen to be the master which assigns password hashes from the password file to the rest of the processors. Each slave processor then runs the entire dictionary against its assigned hash. Upon completion, the slave processor then asks the master for a new hash. The primary advantages to this method are its simplicity and consistent load balancing.

Unfortunately this method is inefficient with real-world workloads. Each processor must keep a private copy of the entire dictionary which can become very memory expensive, especially with large dictionaries (~0.5 Gbyte). Since each processor must run the entire dictionary against each hash assigned to it, this approach may also become computationally expensive with very large dictionaries or when the dictionary is significantly larger than the password file, as is commonly the case.

4. Method

The first phase of this research explored existing MPI implementations of JtR. Several of these implementations were tested and benchmarked for performance. The three primary modes of operation for JtR were analyzed with respect to parallelization. Strategies were also devised for parallelizing each of the three modes, with focus on innovative approaches. As a result, two algorithms were designed to address how to improve the current MPI implementation of JtR.

4.1. Algorithm 1: Distributing the Wordlist Contents amongst Participating Processors

In this algorithm multiple processors are used to crack one password. In this innovative approach the candidate password is distributed to all participating processors and the wordlist is divided based on probability so that each processor has the same likelihood in cracking the password. The overarching goal of this algorithm is to crack a single password as quickly as possible. Figure 1 illustrates the distribution of potential passwords from an original wordlist onto processor 1 to *N*.

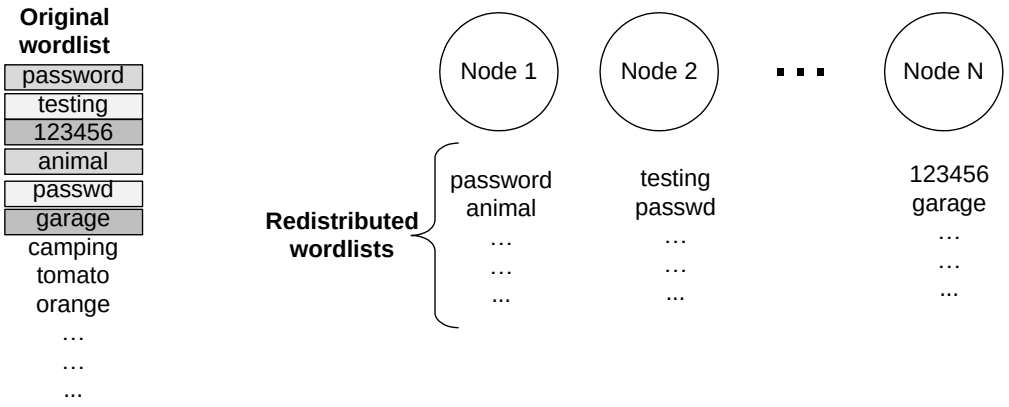

Figure 1. Distributing the wordlist contents fairly amongst participating processors.

The potential benefits of this approach are threefold. One, the most probable passwords are listed at the top of each processor's wordlist thereby increasing the likelihood of finding a match. Two, the wordlist is distributed amongst *N* processors, resulting in gains due to the workload being evenly divided. Three, there is no overlapping of efforts amongst participating processors. In previous versions of JtR, the same passwords are repeatedly cracked by different processors resulting in wasted cpu cycles. Our algorithm ensures that all processors are utilized efficiently collectively attempting to crack the password.

4.2. Algorithm 2: Password file distribution.

The second algorithm is intended to be used by system administrators for systems in which thousands of accounts may be in use. This algorithm divides a potentially large password file equally amongst participating processors. This approach offers efficient load-balancing and fault-tolerant performance. Continual monitoring from the root processor tracks the progress of slave processors in terms of how many passwords have been successfully cracked. If a threshold load factor (λ=50%) is reached, a dynamic and explicit rebalancing occurs in which uncracked passwords are submitted to an underutilized processor. Figure 2 depicts the password distribution used in this algorithm.

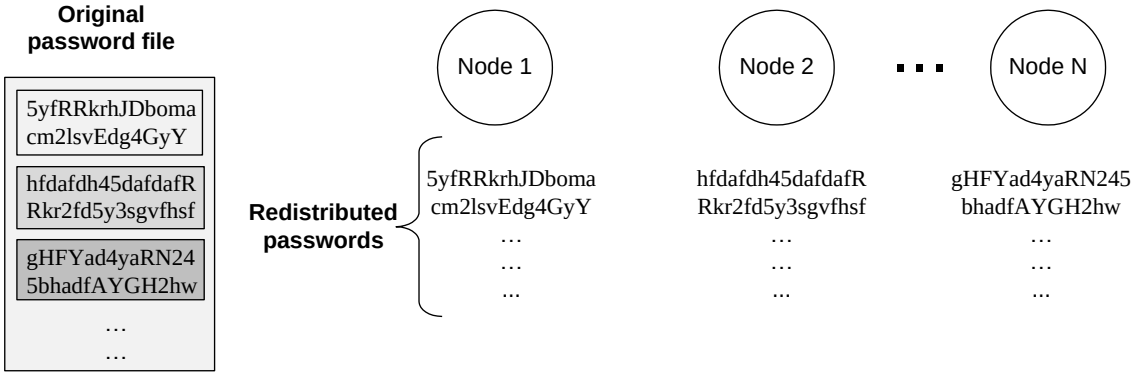

Figure 2. Distributing the password file amongst participating processors while permitting efficient load-balancing.

Benchmarking techniques were used to evaluate the performance of both algorithms. The benchmarks were performed using a known set of passwords encrypted in a chosen format. This password file was subjected to cracking attempts with serial JtR and our MPI implementation, and their results were compared. The dependent variable was the time to crack a given password. The independent variables were the OS, the supercomputer, the supercomputer hardware, the number of processors used, the password hash file and the wordlist. Each benchmark result was averaged over 10 runs to ensure proper benchmarking, repeatability of results, result stability and good result proximity.

5. Preliminary Results

In this section we present the preliminary results of our MPI implementations of Algorithm1 and 2 for JtR. Table 1 shows one of the results obtained from an experiment that evaluated the performance of the Distributed wordlist algorithm. This experiment involved 7 passwords, and a wordlist file that contained 3 million words listed in decreasing probability of likelihood. The program was run on 32 processors on the whale SHARCNET supercomputer[1] [1, 5]. Table 1 shows the time to crack each password for the Original John-MPI and our Distributed wordlist algorithm.

Table 1. 7 password entries in the password file, large wordlist (3 millions words), on 32 processors.

Password	Original John-MPI Time to crack	Distributed worldlist Time to crack
abc123	6 sec	6 sec
guinness	9 min 43 sec	9 min 43 sec
password	6 sec	6 sec
erschlangener	7 days	12 hours
zhora	9 days	3 hours

A second experiment was conducted to evaluate the performance of the original JtR and our distributed password algorithm and to facilitate a comparison between the two. The test was to determine how many passwords could be cracked within a 4 day period using the same supercomputer with 32 processors. A large password file that contained 132 random passwords of variable length up to 14 characters drawn from the full extended ASCII table was used. In this experiment it was found that:

1. the Original JtR cracked 29 of the passwords (21% of total)
2. Algorithm1 (Distributed wordlist), cracked 64 passwords (49% of total)
3. Algorithm 2 (Distributed password) cracked 74 passwords (56% of total)

Table 2 shows these findings.

[1] For details regarding the type of supercomputer used in this research, please see www.sharcnet.ca, specifically: the whale supercomputer: https://www.sharcnet.ca/my/systems/show/27

Table 2. Original JtR, Distributed wordlist algorithm, and Distributed password algorithm experiment findings.

	Number of Passwords Cracked	% of total cracked	Additional passwords cracked
Original JtR	29	21%	---
Algorithm 1: Distributed wordlist	64	49%	35
Algorithm 2: Distributed passwords	74	56%	45

6. Discussion

The research described in this paper focused on enhancements to the wordlist mode of JtR, which uses the dictionary method for cracking passwords. Based on our preliminary findings we have found significant benefits to parallelizing the dictionary cracking method.

Our first algorithm, using a distributed wordlist method, significantly improves the efficiency of password cracking when a large number of processors are available to crack one or two passwords. The distributed wordlist algorithm can crack over twice as many passwords than the original JtR given the same duration of time in which to work. For several tests, this algorithm was shown to be 72 times faster in cracking passwords than the original JtR. Since this method relies on distributing the wordlist file among processors and wordlist contain thousands of entries, its performance should scale well with the addition of more processors.

One of the primary sources of performance improvement over previous implementations is the greater workload distribution. In previous MPI implementations of JtR's wordlist mode, all processors will attempt all entries in the wordlist on the passwords they are assigned. This results in considerable workload duplication in scenarios where the wordlist file is large while the password file is small. The only workaround was to manually abort and resume individual jobs from time to time. Our approach eliminates this workload duplication, making the first algorithm ideal for these scenarios.

Our second algorithm, using a password file distribution method, has also produced promising preliminary results. This algorithm distributes the workload by dividing the password file. In our implementation, processes are working on different password sets, so workload duplication is avoided.

A previous MPI implementation of JtR's wordlist mode [2] similarly divides the workload along the password file, but does so with the addition of inter-process communication; the processors synchronize with a master processor which assigns passwords to each worker processor. Our approach divides the file at the start of runtime which bypasses the overhead of inter-process communication, since the passwords are distributed among the processors at the start of runtime and synchronization between processors is not necessary.

These are preliminary results, and although promising, more tests are needed to determine how effective our algorithms scale, for instance, by using 128, 256 or more processors. Furthermore, experiments involving larger and more varied entries in the password file are needed to draw conclusive findings.

7. Future Research

Future research may focus on combining the two approaches of workload distribution. One possible approach is to use both methods and implement an additional evaluation logic which will determine which method to apply given a specific scenario.

For a scenario where the password list is found to be considerably longer than the number of available processors, the program could use the password distribution technique of algorithm 2. For scenarios where the password list is found to be short, relative to the number of available processors, the second method of workload distribution along the wordlist file implemented in algorithm 1 could be used.

Future work may also explore distributing the password list and the wordlist at the same time. This approach presents a large number of problems involving ensuring proper workload distribution. Any implementation will likely require a high level of inter-process communication and synchronization which will lead to potentially significant overhead.
A new MPI patch was released at near the time of publication of this research {Simpson, 2010 #518}. The patch adds limited workload distribution (auto-splitting in Markov mode), but still does not utilize MPI significantly. MPI is only used to enable each process to know its own relative ID and the total number of processes in a given parallel running group. This implementation should be explored and the inter-process awareness of the distributed wordlist mode can be improved by enabling the processes to notify each other of successful password cracks, therefore avoiding some amount of workload duplication.

Beyond JtR's wordlist mode, future work will include creating an MPI implementation of JtR's incremental mode with a focus on exploring the limitations of current MPI implementations and addressing them. There are no current plans to attempt to parallelize JtR's single crack mode, due to its very short run-time even when run serially. While the single mode can conceivably be parallelized, the benefits of such an endeavour are in doubt.

For incremental mode parallelization, a method of consistent load balancing and processor failure management has yet to be attempted and should be explored. When a processor completes its allocated task, it should be allocated more work taken from the remaining workloads of the other processors. The possible scenario of a processor going offline during execution should also be addressed since execution of incremental mode attacks are very time consuming and graceful processor failure resolution should be explored. This might be implemented by dividing the workload of the failed processor among the remaining processors.

Acknowledgements
This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET:www.sharcnet.ca) and the contributions of my student, Michael Lin, and Sheridan Research member, Wesley Skoczen.

8. References

1. Lim, R., *Parallelization of John the Ripper (JtR) using MPI*. 2004, Computer Science and Engineering University of Nebraska–Lincoln.
2. Pippin, A., B. Hall, and W. Chen, *Parallelization of John the Ripper Using MPI*. 2006, University of California, Santa Barbara.
3. OpenWall (2010) *John the Ripper's cracking modes*.
4. Anderson, J. (2010) *John the Ripper*
5. OpenWall (2010) *MPI with John the Ripper*. 2010.